CHAPTER 12

MOVING INTO IMPLEMENTATION

As the design phase is completed, the systems analyst begins to focus on the tasks associated with building the system, ensuring that it performs as designed and developing documentation for the system. Programmers will carry out the time-consuming and costly task of writing programs, while the systems analyst prepares plans for a variety of tests that will verify that the system performs as expected. Several different types of documentation will also be designed and written during this part of the systems development life cycle.

OBJECTIVES

- Be familiar with the system construction process.
- Explain different types of tests and when to use them.
- Describe how to develop user documentation.

CHAPTER OUTLINE

INTRODUCTION

As the implementation phase begins, foremost on people's minds is *construction* of the new system. A major component of building the system is writing programs. In fact, some people mistakenly believe that programming is the focal point of systems development. We hope you agree that doing a good, thorough job on the analysis and design phases is essential to a smooth and successful implementation phase.

The implementation phase consists of developing and testing the system's software, documentation, and new operating procedures. These topics are presented in this chapter. Chapter 13 discusses additional issues that are essential to a successful system implementation, including installation of the new system, selection of the most suitable conversion approach, preparing the organization and the users to adapt to the new system, and ensuring that the system is supported after it is put into production.

Developing the system's software (writing programs) can be the largest single component of any systems development project in terms of both time and cost. It is generally also the best understood component and may offer the fewest problems of all the aspects of the SDLC. Since the systems analyst is usually not actually doing the programming (programmers are), in this chapter we concentrate our attention on managing the programming process.

While programmers are transforming program specifications into working program code, the systems analysts will be designing a variety of tests that will be performed on the new system. As the programs are finalized, the systems analysts may conduct these tests to verify that the system actually does what it was designed to do. Testing may be a major element of the implementation phase for the systems analysts. (In some organizations, testing is performed by specialized quality assurance personnel.)

During this phase, it is also the responsibility of the systems analysts to finalize the system documentation and develop the user documentation. The final section of this chapter discusses the various types of documentation that must be prepared.

MANAGING THE PROGRAMMING PROCESS

The programming process is quite well understood and generally flows smoothly. When system development projects fail, it is usually not because the programmers were unable to write the programs. Flaws in analysis, design, or project management are the leading contributors to project failure. In order to ensure that the process of programming is conducted successfully, we discuss several tasks that the project manager must do to manage the programming effort: assigning programming tasks, coordinating the activities, and managing the programming schedule.[1]

Assigning Programming Tasks

During project planning (Chapter 2), the project manager identified the programming support required for constructing the system in terms of the numbers and skill levels of programmers. Now the project manager must assign program modules to the programming staff. As discussed in Chapter 10, each programming module should be as separate and distinct as possible from the other modules. The project

[1] One of the best books on managing programming (even though it was first written in 1975) is that by Frederick P. Brooks Jr. *The Mythical Man-Month*, 20th anniversary ed., Reading, MA: Addison-Wesley, 1995.

CONCEPTS IN ACTION

12-A The Cost of a Bug

My first programming job in 1977 was to convert a set of application systems from one version of COBOL to another version of COBOL for the government of Prince Edward Island. The testing approach was to first run a set of test data through the old system and then run it through the new system to ensure that the results from the two matched. If they matched, then the last three months of production data were run through both to ensure they, too, matched.

Things went well until I began to convert the gas tax system that kept records on everyone authorized to purchase gasoline without paying tax. The test data ran fine, but the results using the production data were peculiar. The old and new systems matched, but rather than listing several thousand records, the report listed only 50. I checked the production data file and found it listed only 50 records, not the thousands that were supposed to be there.

The system worked by copying the existing gas tax records file into a new file and making changes in the new file. The old file was then copied to tape backup. There was a bug in the program such that if there were no changes to the file, a new file was created, but no records were copied into it.

I checked the tape backups and found one with the full set of data that was scheduled to be overwritten three days after I discovered the problem. The government was only three days away from losing all gas tax records.

Alan Dennis

Question:

What might have happened if this bug hadn't been caught and all gas tax records were lost?

manager first groups together modules that are related. These groups of modules are then assigned to programmers on the basis of their experience and skill level. Experienced, skilled programmers will be assigned the most complex modules, while novice programmers will be given less complex ones.

It is quite likely that there will be a mismatch between the available programming skills and the programming skills that are needed to complete the programming. Consequently, the project manager must take steps at this time to ensure that skill deficiencies are eliminated through additional training or through mentoring arrangements with more experienced, skilled programmers. When the required skills are not readily available, the project manager must recognize the need for additional time in the project schedule.

While it will be tempting to speed up the programming process by adding more programming staff to the project, an ironic fact of system development is that the more programmers who are involved, the longer the project will take. As the size of the programming team increases, the need for coordination increases exponentially, and the more coordination that is required, the less time programmers can spend actually writing programs. The best size is the smallest feasible programming team. When projects are so complex that they require a large team, the best strategy is to try to break the project into a series of smaller parts that can function as independently as possible.

Coordinating Activities

Coordination can be done through both high-tech and low-tech means. The simplest approach is to have a weekly project meeting to discuss any changes to the system that have arisen during the past week—or just any issues that have come up. Regular meetings, even if they are brief, encourage the widespread communication and discussion of issues before they become problems.

Another important way to improve coordination is to create and follow standards that can range from formal rules for naming files to forms that must be completed when goals are reached to programming guidelines. (See Chapter 2.) When a team forms standards and then follows them, the project can be completed faster because task coordination is less complex.

The project manager must put mechanisms in place to keep the programming effort well organized. Many project teams set up three "areas" in which programmers can work: a development area, a testing area, and a production area. These areas can be different directories on a server hard disk, different servers, or different physical locations, but the point is that files, data, and programs are separated on the basis of their status of completion. At first, programmers access and build files within the development area. Then they copy them to the testing area when they are "finished." If a program does not pass a test, it is sent back to development. Once all the programs are tested and ready to support the new system, they are copied into the production area—the location where the final system will reside.

Keeping files and programs in different places according to completion status helps manage *change control*, the action of coordinating a program as it changes through construction. Another change control technique is keeping track of what programs are being changed by whom, through the use of a *program log*. The log is merely a form on which programmers sign out programs to write, and sign in the programs when they are completed. Both the programming areas and program log help the analysts understand exactly who has worked on what and the program's status. Without these techniques, files can be put into production without the proper testing, two programmers can start working on the same program at the same time, files can be overlooked, and so on. Code management systems are available that facilitate the "checkout" of programs and maintain various versions of a module.

Many CASE tools are set up to track the status of programs and help manage programmers as they work. In most cases, maintaining coordination is not conceptually complex. It just requires a lot of attention and discipline to track small details.

Managing the Schedule

The time estimates that were produced during the initial planning phase and refined during the analysis and design phases must almost always be refined as the project progresses during construction, because it is virtually impossible to develop an exact assessment of the project's schedule. As we discussed in Chapter 2, a well-done set of time estimates will usually have a 10% margin of error by the time implementation is reached. It is critical that the time estimates be revised as the construction step proceeds. If a program module takes longer to develop than expected, then the prudent response is to move the expected completion date later by the same amount of time.

One of the most common causes for schedule problems is *scope creep*. Scope creep occurs when new requirements are added to the project after the system design has been finalized. Scope creep can be very expensive because changes made late in the SDLC can require much of the completed system design (and even programs already written) to be redone. Any proposed change during construction will require the approval of the project manager and should be addressed only after a quick cost–benefit analysis has been done.

Another common cause is the unnoticed day-by-day slippages in the schedule. One module is a day late here; another one, a day late there. Pretty soon these minor delays add up, and the project is noticeably behind schedule. Once again, the key to managing the programming effort is to watch these minor slippages carefully and update the schedule accordingly. It is especially critical to monitor slippage of all tasks on the critical path, since falling behind on these tasks will affect the final completion date for the project.

Typically, a project manager will create a risk assessment that tracks potential risks, along with an evaluation of their likelihood and potential impact. As programming progresses, the list of risks will change as some items are removed and others surface. The best project managers, however, work hard to keep risks from having an impact on the schedule and costs associated with the project.

TESTING

Writing programs is a fun, creative activity. Novice programmers tend to get caught up in the development of the programs themselves and are often much less enchanted with the tasks of testing and documenting their work. Testing and documentation aren't fun; consequently, they receive less attention than writing the programs.

PRACTICAL TIP

12-1 Avoiding Classic Implementation Mistakes

In previous chapters, we discussed classic mistakes and how to avoid them. Here, we summarize four classic mistakes in the implementation phase:

1. **Research-oriented development:** Using state-of-the-art technology requires research-oriented development that explores the new technology, because "bleeding-edge" tools and techniques are not well understood, are not well documented, and do not function exactly as promised.
   Solution: If you use state-of-the-art technology, you should significantly increase the project's time and cost estimates even if (some experts would say *especially if*) such technologies claim to reduce time and effort.
2. **Using low-cost personnel:** You get what you pay for. The lowest-cost consultant or staff member is significantly less productive than the best staff. Several studies have shown that the best programmers produce software six to eight times faster than the least productive (yet cost only 50% to 100% more).
   Solution: If cost is a critical issue, assign the best, most expensive personnel; never assign entry-level personnel in an attempt to save costs.
3. **Lack of code control:** On large projects, programmers must coordinate changes to the program source code (so that two programmers don't try to change the same program at the same time and one doesn't overwrite the other's changes). Although manual procedures appear to work (e.g., sending e-mail notes to others when you work on a program to tell them not to work on that program), mistakes are inevitable.
   Solution: Use a source code library that requires programmers to check out programs and prohibits others from working on them at the same time.
4. **Inadequate testing:** The number-one reason for project failure during implementation is ad hoc testing—in which programmers and analysts test the system without formal test plans.
   Solution: Always allocate sufficient time in the project plan for formal testing.

Source: Adapted from *Rapid Development*, Redmond, WA: Microsoft Press, 1996, pp. 29–50, by Steve McConnell.

Programming and testing are very similar to writing and editing, however. No professional writer (or serious student writing an important term paper) would stop after writing the first draft. Rereading, editing, and revising the initial draft into a good paper is the hallmark of good writing. Likewise, thorough testing is the hallmark of professional software developers. Most professional organizations devote more time and money to testing (and to revision and retesting) than to writing the programs in the first place.

The attention paid to testing is justified by the high costs associated with downtime and failures caused by software bugs.[2] Software bugs are estimated to cost the U.S. economy $59.5 billion annually.[3] One serious bug that causes an hour of downtime can cost more than one year's salary of a programmer—and how often are bugs found and fixed in an hour? Testing is therefore a form of insurance. Organizations are willing to spend a lot of time and money to prevent the possibility of major failures after the system is installed. Figure 12-1 lists some estimated income losses for several industries that cannot function without their computer systems.

A program is not considered finished until it has passed its testing. For this reason, programming and testing are tightly coupled. Testing is frequently the primary focus of the systems analysts as the system is being constructed. The analysts must resist the temptation to rush into testing as soon as the very first program module is complete, however. Spontaneously testing different events and possibilities without spending time to develop a comprehensive test plan is dangerous, because important tests may be overlooked. If an error does occur, it may be difficult to reproduce the exact sequence of events that cause it. Instead, testing must be performed and documented systematically so that the project team always knows what has and has not been tested.

FIGURE 12-1
Estimated Lost Income Resulting from One Hour of System Downtime, By Industry

Brokerage Service	$6.4 million
Energy	2.8 million
Telecom	2.0 million
Manufacturing	1.6 million
Retail	1.1 million
Health Care	636,000
Media	90,000
"Assessing the Financial Impact of Downtime," Vision Solutions, 2008, www.visionsolutions.com	

[2] When I was an undergraduate, I had the opportunity to hear Admiral Grace Hopper tell how the term *bug* was introduced. She was working on one of the early U.S. Navy computers when suddenly it failed. The computer would not restart properly, so she began to search for failed vacuum tubes. She found a moth inside one tube and recorded in the log book that a bug had caused the computer to crash. From then on, every computer crash was jokingly blamed on a bug (as opposed to programmer error), and eventually the term *bug* entered the general language of computing.

[3] See "Software Errors Cost U.S. Economy $59.5 Billion Annually," NIST Report 2002–10, www.nist.gov/public_affairs/releases/no2-10.htm.

The sections that follow describe a number of different types of tests that must be performed prior to installing the new system. Each type of test checks different features and/or scope of the system, until ultimately it is tested for acceptance by the users.

Test Planning

Testing starts with the tester's developing a *test plan* that defines a series of tests that will be conducted.[4] Figure 12-2 shows a typical test plan form. A test plan often has 20 to 30 pages, with a separate page for each individual test in the plan. Each individual test has a specific objective, describes a set of very specific *test cases* to examine, and defines the expected results and the actual results observed. The test objective is taken directly from the program specification or from the program source code. For example, suppose that the program specification stated that the order quantity must be between 10 and 100 cases. The tester would develop a series of test cases to ensure that the quantity is validated before the system accepts it.

It is impossible to test every possible combination of input and situation; there are simply too many possible combinations. In this example of an order quantity that must be between 10 and 100 cases, the test requires a minimum of 3 test cases: one with a valid value (e.g., 15), one with an invalid value too low (e.g., 7), and one with an invalid value too high (e.g., 110). Most tests would also include a test case with a non-numeric value to ensure that the data types were checked (e.g., ABCD). A really good test would include a test case with nonsensical, but potentially valid, data (e.g., 21.4).

In some cases, test cases cannot be conducted by entering data values, but must instead be handled by selecting certain combinations of commands or menu choices. The script area on the test plan is used to describe the sequence of keystrokes or mouse clicks and movements for this type of test.

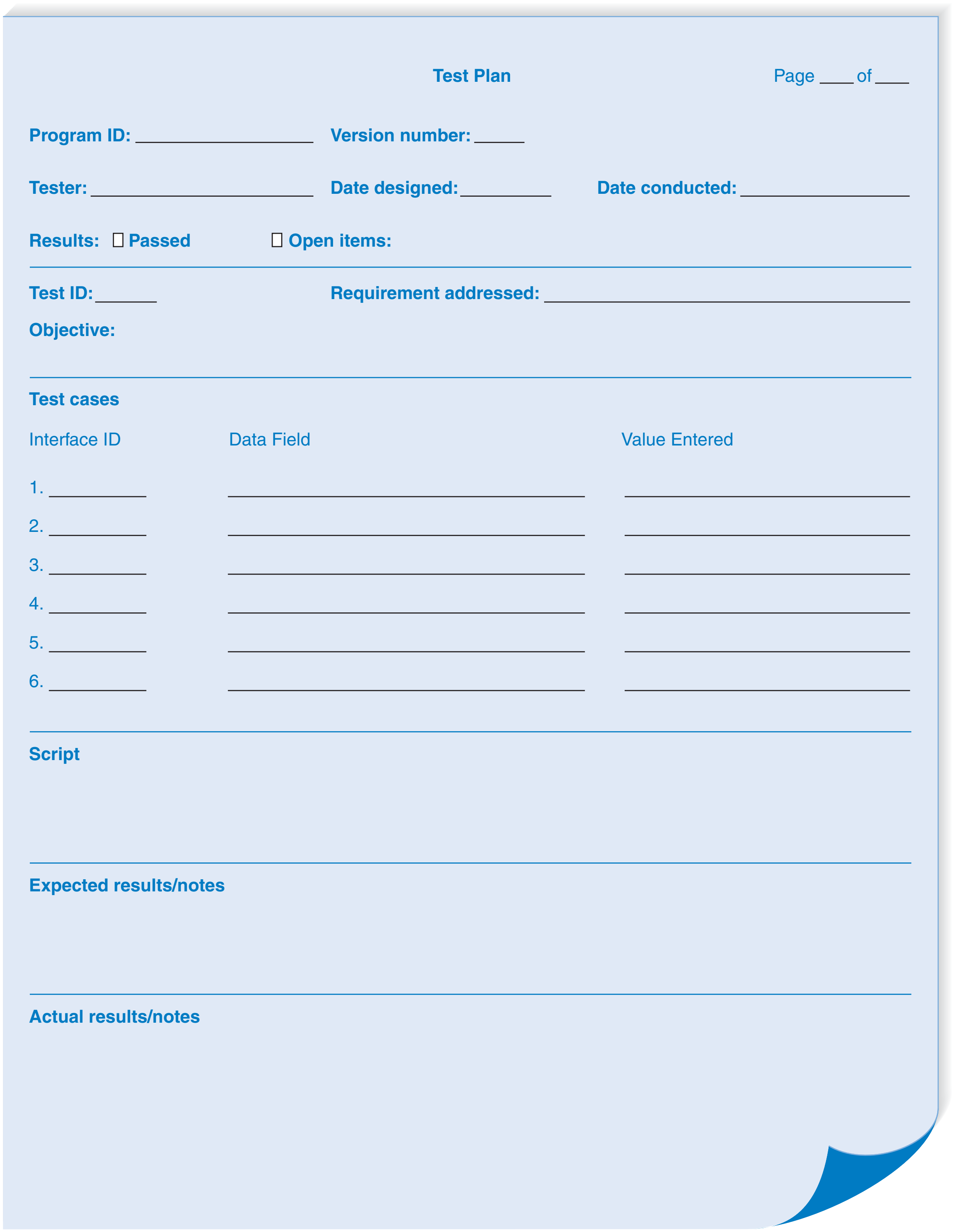
Not all program modules are likely to be finished at the same time, so the programmer usually writes *stubs* for the unfinished modules to enable the modules around them to be tested. A stub is a placeholder for a module that usually displays a simple test message on the screen or returns some *hardcoded* value[5] when it is selected. For example, consider an application system that provides the five standard functions discussed in Chapter 5 for some data objects such as customers, vehicles, or employees: creating, changing, deleting, finding, and printing (whether on the screen or on a printer). Each of these functions could be a separate module that needs to be tested, and in fact, printing might be two separate modules, one for an on-screen list and one for the printer (Figure 12-3).

Suppose that the main menu module in Figure 12-3 was complete. It would be impossible to test it properly without the other modules, because the function of the main menu is to navigate to the other modules. In this case, a stub would be written for each of the other modules. These stubs would simply display a message on the screen when they were activated (e.g., "Delete item module reached"). In this way, the main menu module could pass module testing before the other modules were completed.

There are four general stages of tests: unit tests, integration tests, system tests, and acceptance tests. Although each application system is different, most errors are found during integration and system testing (Figure 12-4).

[4] For more information on testing, see William Perry, *Effective Methods for Software Testing*, "3d" ed. 2006.

[5] The word *hardcoded* means "written into the program." For example, suppose that you were writing a unit to calculate the net present value of a loan. The stub might be written to always display (or return to the calling module) a value of 100 regardless of the input values. In this case, we would say that the 100 was hardcoded.



Test Plan Page ____ of ____

Program ID: ____________________ **Version number:** ______

Tester: ________________________ **Date designed:** __________ **Date conducted:** __________________

Results: ☐ **Passed** ☐ **Open items:**

Test ID: ______ **Requirement addressed:** ____

Objective:

Test cases

Interface ID	Data Field	Value Entered
1. __________	________________________________	__________________________
2. __________	________________________________	__________________________
3. __________	________________________________	__________________________
4. __________	________________________________	__________________________
5. __________	________________________________	__________________________
6. __________	________________________________	__________________________

Script

Expected results/notes

Actual results/notes

FIGURE 12-2
Test Plan

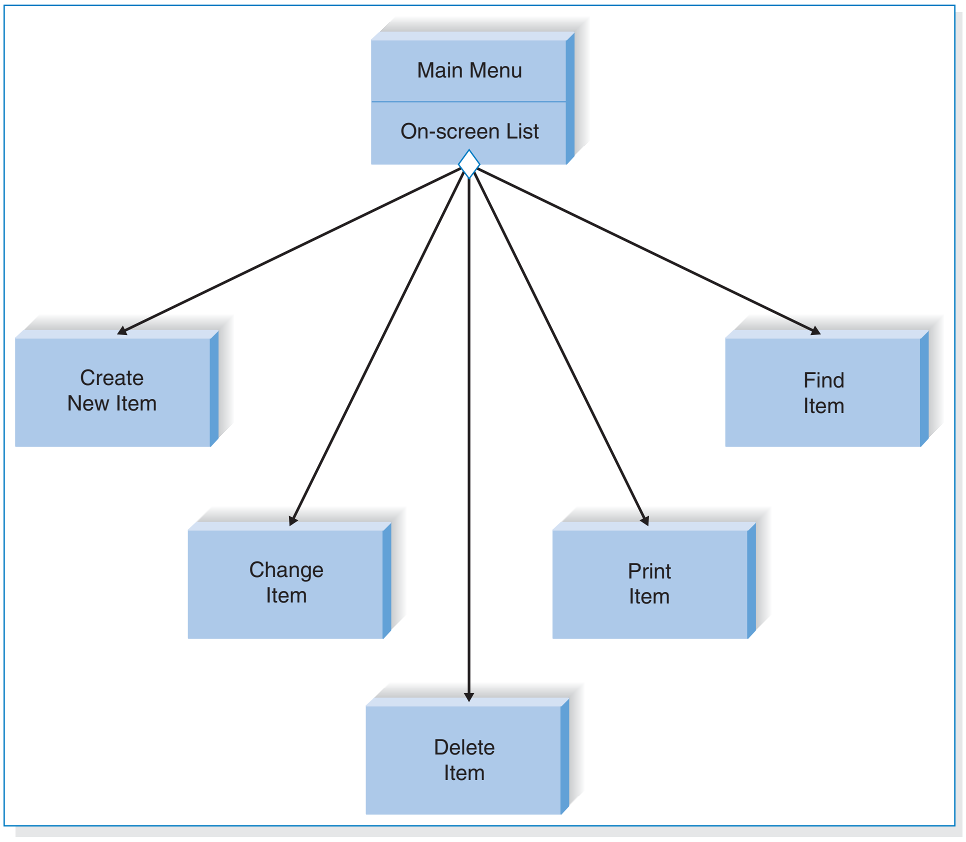


FIGURE 12-3
Testing Separate Modules

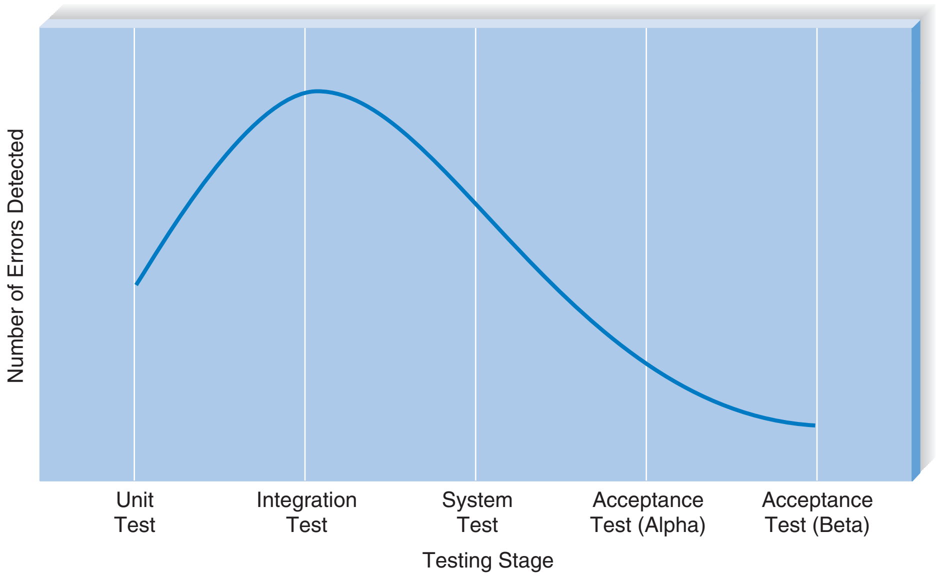


FIGURE 12-4
Error Discovery Rates for Different Stages of Tests

YOUR TURN

12-1 Test Planning for an Automated Teller Machine

Pretend that you are a project manager for a bank developing software for automated teller machines (ATMs). Develop a unit test plan for the user interface component of the ATM.

Unit Tests

Unit tests focus on one unit—a program or a program module that performs a specific function that can be tested. The purpose of a unit test is to ensure that the module or program performs its function as defined in the program specification. Unit testing is performed after the programmer has developed and tested the code and believes it to be error free. These tests are based strictly on the program specification and may discover errors resulting from the programmer's misinterpretation of the specifications. Unit tests are often conducted by the systems analyst or, sometimes, by the programmer who developed the unit.

There are two approaches to unit testing: *black-box* and *white-box* (Figure 12-5). Black-box testing is the most commonly used. In this case, the test plan is developed directly from the program specification: Each item in the program specification becomes a test, and several test cases are developed for it. White-box testing is reserved for special circumstances in which the tester wants to review the actual program code, usually when complexity is high.

Integration Tests

Integration tests assess whether a set of modules or programs that must work together do so without error. They ensure that the interfaces and linkages between different parts of the system work properly. At this point, the modules have passed their individual unit tests, so the focus now is on the flow of control among modules and on the data exchanged among them. Integration testing follows the same general procedures as unit testing: the tester develops a test plan that has a series of tests. Integration testing is often done by a set of programmers and/or systems analysts.

There are four approaches to integration testing: user interface testing, use scenario testing, data flow testing, and system interface testing. (See Figure 12-5.) Most projects use all four approaches.

System Tests

System tests are usually conducted by the systems analysts to ensure that all modules and programs work together without error. System testing is similar to integration testing, but is much broader in scope. Whereas integration testing focuses on whether the modules work together without error, system tests examine how well the system meets business requirements and its usability, security, and performance under heavy load (see Figure 12-5). It also tests the system's documentation.

Stage	Types of Tests	Test Plan Source	When to Use	Notes
Unit Testing	**Black-box testing:** treats program as black box.	Program specifications	For normal unit testing	The tester focuses on whether the unit meets the requirements stated in the program specifications.
	White-box testing: looks inside the program to test its major elements.	Program source code	When complexity is high	By looking inside the unit to review the code itself, the tester may discover errors or assumptions not immediately obvious to someone treating the unit as a black box.
Integration Testing	**User interface testing:** The tester tests each interface function.	Interface design	For normal integration testing	Testing is done by moving through each and every menu item in the interface either in a top-down or bottom-up manner.
	Use scenario testing: The tester tests each use scenario.	Use scenario	When the user interface is important	Testing is done by moving through each use scenario to ensure that it works correctly. Use scenario testing is usually combined with user interface testing because it does not test all interfaces.
	Data flow testing: Tests each process in a step-by-step fashion.	Physical DFDs	When the system performs data processing	The entire system begins as a set of stubs. Each unit is added in turn, and the results of the unit are compared with the correct result from the test data; when a unit passes, the next unit is added and the test is rerun.
	System interface testing: tests the exchange of data with other systems.	Physical DFDs	When the system exchanges data	Because data transfers between systems are often automated and not monitored directly by the users, it is critical to design tests to ensure that they are being done correctly.
System Testing	**Requirements testing:** tests whether original business requirements are met.	System design, unit tests, and integration tests	For normal system testing	This test ensures that changes made as a result of integration testing did not create new errors. Testers often pretend to be uninformed users and perform improper actions to ensure that the system is immune to invalid actions (e.g., adding blank records).
	Usability testing: tests how convenient the system is to use.	Interface design and use scenarios	When user interface is important	This test is often done by analysts with experience in how users think and in good interface design. This test sometimes uses the formal usability testing procedures discussed in Chapter 9.
	Security testing: tests disaster recovery and unauthorized access.	Infrastructure design	When the system is important	Security testing is a complex task, usually done by an infrastructure analyst assigned to the project. In extreme cases, a professional firm may be hired.
	Performance testing: examines the ability to perform under high loads.	System proposal and infrastructure design	When the system is important	High volumes of transactions are generated and given to the system. This test is often done by the use of special-purpose testing software.
	Documentation testing: tests the accuracy of the documentation.	Help system, procedures, tutorials	For normal system testing	Analysts spot-check or check every item on every page in all documentation to ensure that the documentation items and examples work properly.
Acceptance Testing	**Alpha testing:** conducted by users to ensure that they accept the system.	System tests	For normal acceptance testing	Alpha tests often repeat previous tests, but are conducted by users themselves to ensure that they accept the system.
	Beta testing: uses real data, not test data.	No plan	When the system is important	Users closely monitor the system for errors or useful improvements.

DFD = data flow diagram.

FIGURE 12-5
Types of Tests

Acceptance Tests

Acceptance tests are done primarily by the users with support from the project team. The goal is to confirm that the system is complete, meets the business needs that prompted the system to be developed, and is acceptable to the users. Acceptance testing is done in two stages: *alpha testing*, in which users test the system using made-up data, and *beta testing*, in which users begin to use the system with real data and carefully monitor the system for errors. (See Figure 12-5.)

The users' perceptions of the new system will be significantly influenced by their experiences during acceptance testing. Since first impressions are sometimes difficult to change, analysts should strive to ensure that acceptance testing is conducted only following rigorous (and successful) system testing. In addition, listening to and responding to user feedback will be essential in shaping a positive reaction to and acceptance of the new system by the users.

DEVELOPING DOCUMENTATION

There are two fundamentally different types of documentation. *System documentation* is intended to help programmers and systems analysts understand the application software and enable them to build it or maintain it after the system is installed. System documentation is a by-product of the systems analysis and design process and is created as the project unfolds. Each step and phase produces documents that are essential in understanding how the system is built or is to be built, and these documents are stored in the project binder(s).

User documentation (such as user manuals, training manuals, and online help systems) is designed to help the user operate the system. Although most project teams expect users to have received training and to have read the user manuals before operating the system, unfortunately, this is not always the case. It is more common today—especially in the case of commercial software packages

CONCEPTS IN ACTION

12-B Managing a Database Project

A consulting project involved a credit card "bottom feeder" (let's call it Credit Wonder). This company bought credit card accounts that were written off as uncollectable debts by major banks. Credit Wonder would buy the write-off accounts for 1 or 2 percent of their value and then would call the owners of the written-off accounts and "offer a deal" to the credit card account holders.

Credit Wonder wanted a database for these accounts. Legally, it did own them and so could contact the people who had owed the money—but as prescribed in credit law. For example, Credit Wonder could call only during certain hours and no more than once a week, and they had to speak to the actual account holder. Any amount collected over the 1 to 2 percent of the original debt would be considered a gain. In its database, Credit Wonder wanted a history of what settlement was offered, the date the account holder was contacted, and additional notes.

Questions:

1. How might a systems analyst manage such a system project?
2. Who would the systems analyst need to interview to get the system requirements?
3. How would a database analyst help in structuring the database requirements?

for microcomputers—for users to begin using the software without training or reading the user manuals. In this section, we focus on user documentation.[6]

User documentation is often left until the end of the project, which is a dangerous strategy. Developing good documentation takes longer than many people expect, because it requires much more than simply writing a few pages. Producing documentation requires designing the documents (whether paper or online), writing the text, editing them, and testing them. For good-quality documentation, this process usually takes about 3 hours per page (single-spaced) for paper-based documentation or 2 hours per screen for online documentation. Thus, a "simple" set of documentation such as a 10-page user manual and a set of 20 help screens takes 70 hours. Of course, lower-quality documentation can be produced faster.

The time required to develop and test user documentation should be built into the project plan. Most organizations plan for documentation development to start once the interface design and program specifications are complete. The initial draft of documentation is usually scheduled for completion immediately after the unit tests are complete. This reduces—but doesn't eliminate—the chance that the documentation will need to be changed because of software changes, and it still leaves enough time for the documentation to be tested and revised before the acceptance tests are started.

Although paper-based manuals are still important, online documentation is becoming the predominant form. Paper-based documentation is simpler to use because it is more familiar to users, especially novices who have less computer experience; online documentation requires the users to learn one more set of commands. Paper-based documentation also is easier to flip through to gain a general understanding of its organization and topics and can be used far away from the computer itself.

There are four key strengths of online documentation, however, which all but guarantee its position as the dominant form for the foreseeable future. First, searching for information is often simpler (provided that the help search index is well designed). The user can type in a variety of keywords to view information almost instantaneously, rather than having to search through the index or table of contents in a paper document. Second, the same information can be presented several times in many different formats, so that the user can find and read the information in the most informative way. (Such redundancy is possible in paper documentation, but the cost and intimidating size of the resulting manual make it impractical.) Third, online documentation enables the user to interact with the documentation in many new ways that are not possible with static paper documentation. For example, it is possible to use links or "tool tips" (i.e., pop-up text; see Chapter 9) to explain unfamiliar terms, and programmers can write "show me" routines that demonstrate on the screen exactly what buttons to click and what text to type. Finally, online documentation is significantly less expensive to distribute and keep up to date than paper documentation.

Types of Documentation

There are three fundamentally different types of user documentation: reference documents, procedures manuals, and tutorials. *Reference documents* (also called the help system) are designed to be used when the user needs to learn how to

[6] For more information on developing documentation, see Thomas T. Barker, *Writing Software Documentation*, Boston: Allyn & Bacon, 1998.

perform a specific function (e.g., updating a field, adding a new record). Typically, people read reference information only after they have tried and failed to perform the function. Writing reference documents requires special care because users are often impatient or frustrated when they begin to read them.

Procedures manuals describe how to perform business tasks (e.g., printing a monthly report, taking a customer order). Each item in the procedures manual typically guides the user through a task that requires several functions or steps in the system. Therefore, each entry is typically much longer than an entry in a reference document.

Tutorials teach people how to use major components of the system (e.g., an introduction to the basic operations of the system). Each entry in the tutorial is typically longer still than the entries in procedures manuals, and the entries are usually designed to be read in sequence, whereas entries in reference documents and procedures manuals are designed to be read individually.

Regardless of the type of user documentation, the overall process for developing it is similar to the process of developing interfaces (see Chapter 9). The developer first designs the general structure for the documentation and then develops the individual components within it.

Designing Documentation Structure

In this section, we focus on the development of online documentation because we believe that it is the most common form of user documentation. The general structure used in most online documentation, whether reference documents, procedures manuals, or tutorials, is to develop a set of *documentation navigation controls* that lead the user to *documentation topics.* The documentation topics are the material that users want to read, whereas the navigation controls are the way in which users locate and access a specific topic.

Designing the structure of the documentation begins by identifying the different types of topics and navigation controls that must be included. Figure 12-6 shows a commonly used structure for online reference documents (i.e., the help system). The documentation topics generally come from three sources. The first and most obvious source of topics is the set of commands and menus in the user interface. This set of topics is very useful if the user wants to understand how a particular command or menu is used.

However, users often don't know what commands to look for or where they are in the system's menu structure. Instead, users have tasks they want to perform, and rather than thinking in terms of commands, they think in terms of their business tasks. Therefore, the second and often more useful set of topics focuses on how to perform certain tasks, usually those in the use scenarios from the user interface design. (See Chapter 9.) These topics walk the user through the set of steps (often involving several keystrokes or mouse clicks) needed to perform some task.

The third set of topics are definitions of important terms. These terms are usually the entities and data elements in the system, but sometimes they also include commands.

There are five general types of navigation controls for topics, but not all systems use all five types. (See Figure 12-6.) The first is the table of contents that organizes the information in a logical form, as though the users were to read the reference documentation from start to finish. The second, the index, provides access into the topics via important keywords, in the same way that the index at the back

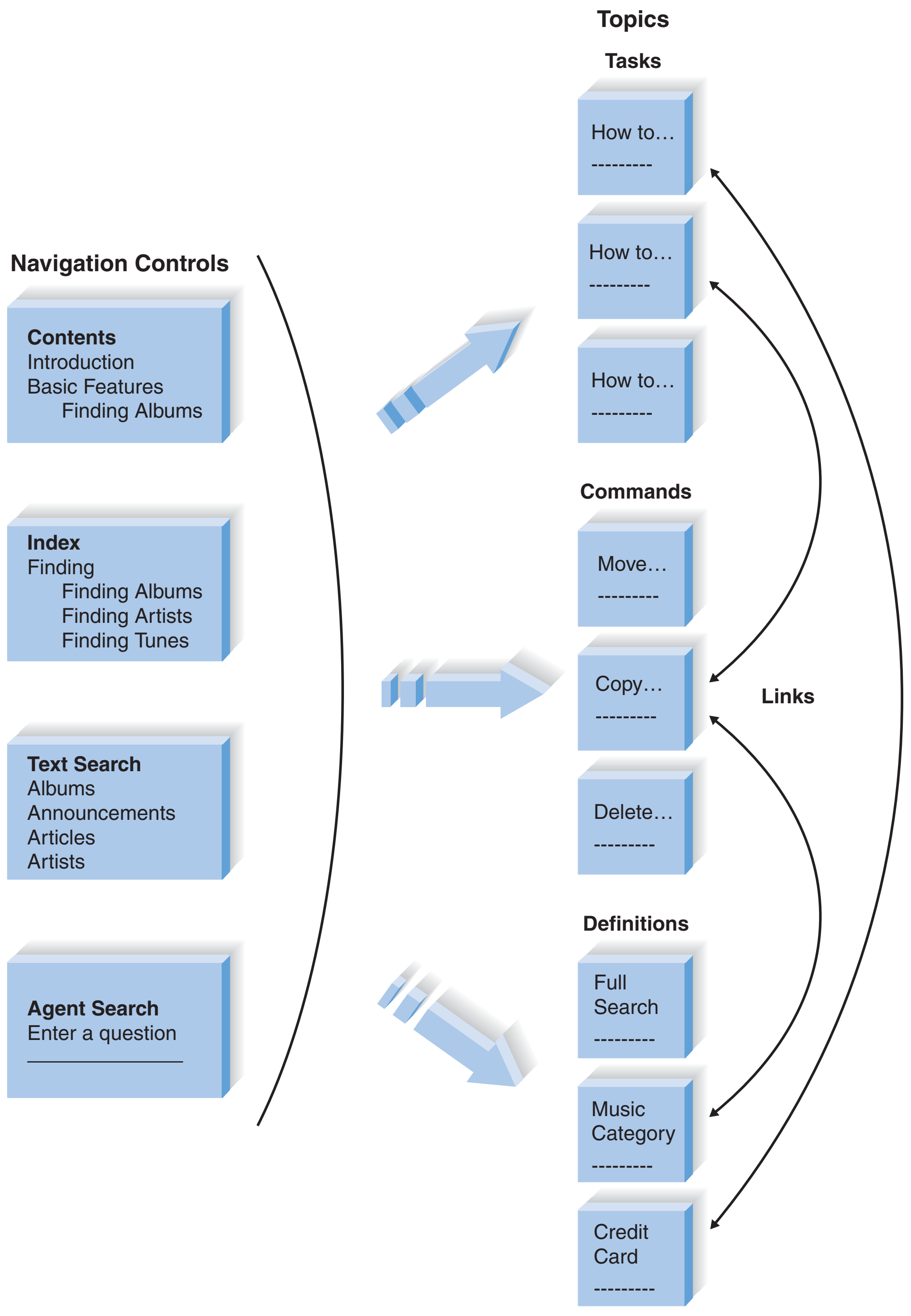


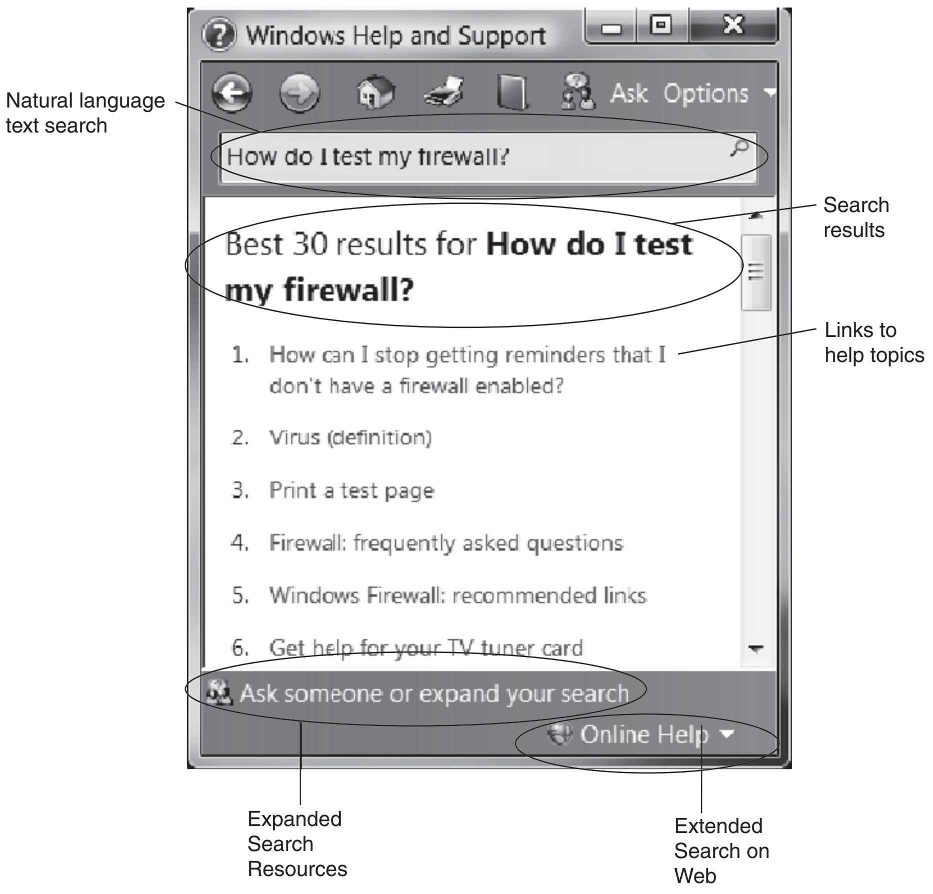
FIGURE 12-6
Organizing Online Reference Documents

of a book helps you to find topics. Third, text search provides the ability to search through the topics either for any text the user types or for words that match a developer-specified set of words that is much larger than the list of words in the index. Unlike the index, text search typically provides no organization to the words (other than alphabetic). Fourth, some systems provide the ability to use an intelligent agent to help in the search. The fifth and final navigation control to topics are the Web-like links between topics that enable the user to click and move among topics.

Procedure manuals and tutorials are similar, but often simpler in structure. When the new system significantly changes the way things are done, these resources are very important. Topics for procedures manuals usually come from the use scenarios developed during interface design and from other basic tasks the users must perform. Topics for tutorials are usually organized around major sections of the system and the level of experience of the user. Most tutorials start with basic, most commonly used commands and then move into more complex and less frequently used commands.

Writing Documentation Topics

The general format for topics is fairly similar across application systems and operating systems (Figure 12-7). Topics typically start with very clear titles, followed by some introductory text that defines the topic, and then provide detailed, step-by-step instructions on how to perform what is being described (where appropriate). Many topics include screen images to help the user find items on the screen; some also have "show me" examples in which the series of keystrokes and/or mouse movements and clicks needed to perform the function


Windows Help and Support
Ask
Options
Natural language text search
How do I test my firewall?
Search results
Links to help topics
Ask someone or expand your search
Online Help
Expanded Search Resources
Extended Search on Web

Best 30 results for **How do I test my firewall?**

1. How can I stop getting reminders that I don't have a firewall enabled?
2. Virus (definition)
3. Print a test page
4. Firewall: frequently asked questions
5. Windows Firewall: recommended links
6. Get help for your TV tuner card

FIGURE 12-7
A Help Topic in Microsoft Windows

YOUR TURN

12-2 Documentation for an Automated Teller

Pretend that you are a project manager for a bank, developing software for automated teller machines. Develop an online help system.

are demonstrated to the user. Most also include navigation controls to enable movement among topics, usually at the top of the window, plus links to other topics. Some also have links to related topics that include options or other commands and tasks the user may want to perform in concert with the topic being read.

Writing the topic content can be challenging. It requires a good understanding of the users (or, more accurately, the range of users) and a knowledge of what skills the users currently have and can be expected to import from other systems and tools they are using or have used (including the system the new system is replacing). Topics should always be written from the viewpoint of the user and describe what the user wants to accomplish, not what the system can do. Figure 12-8 provides some general guidelines to improve the quality of documentation text.[7]

Identifying Navigation Terms

As you write the documentation topics, you also begin to identify the terms that will be used to help users find topics. The table of contents is usually the most straightforward, because it is developed from the logical structure of the documentation topics, whether reference topics, procedure topics, or tutorial topics. The items for the index and search engine require more care because they are developed from the major parts of the system and the users' business functions. Every time you write a topic, you must also list the terms that will be used to find the topic. Terms for the index and search engine can come from four distinct sources.

The first source for index terms is the set of the commands in the user interface, such as *open file*, *modify customer*, and *print open orders.* All commands contain two parts (action and object). It is important to develop the index for both parts because users could search for information by using either part. A user looking for more information about saving files, for example, might search by using the term *save* or the term *files.*

The second source is the set of major concepts in the system, which are often the entities, data stores, and data elements in the data flow diagrams. In the case of Tune Source, for example, this might include *music genre*, *artist*, and *tune.*

A third source is the set of business tasks the user performs, such as ordering replacement units or making an appointment. Often these will be contained in the

[7] One of the best books to explain the art of writing is that by William Strunk, Jr., and E. B. White, *Elements of Style*, 3d ed., Needham Heights, MA: Allyn & Bacon, 1995.

Guideline	Before the Guideline	After the Guideline
Use the active voice: The active voice creates more active and readable text by putting the subject at the start of the sentence, the verb in the middle, and the object at the end.	Finding tunes is done by using the tune title, the artist's name, or a genre of music.	Find a tune by the tune title, the artist's name, or a music genre.
Use e-prime style: E-prime style creates more active writing by omitting all forms of the verb *to be*.	The text you want to copy must be selected before you click on the *copy* button.	Select the text you want to copy before you click on the *copy* button.
Use consistent terms: Always use the same term to refer to the same items, rather than switching among synonyms (e.g., change, modify, update).	Select the text you want to copy. Pressing the *copy* button will copy the marked text to the new location.	Select the text you want to copy. Press the copy button to copy the selected text. Press the *paste* button to place the text into the new location.
Use simple language: Always use the simplest language possible to accurately convey the meaning. This does not mean that you should "dumb down" the text, but that you should avoid artificially inflating its complexity. Avoid separating subjects and verbs and try to use the fewest words possible. (When you encounter a complex piece of text, try eliminating words; you may be surprised at how few words are really needed to convey meaning.)	The Georgia Statewide Academic and Medical System (GSAMS) is a cooperative and collaborative distance learning network in the state of Georgia. The organization in Atlanta that administers and manages the technical and overall operations of the currently more than 300 interactive audio and video teleconferencing classrooms throughout the Georgia system is the Department of Administrative Service (DOAS). (56 words)	The Department of Administrative Service (DOAS) in Atlanta manages the Georgia Statewide Academic and Medical System (GSAMS), a distance learning network with more than 300 teleconferencing classrooms throughout Georgia. (29 words)
Use friendly language: Too often, documentation is cold and sterile because it is written in a very formal manner. Remember, you are writing for a person, not a computer.	Blank disks have been provided to you by the operations department. It is suggested that you make backup copies of all essential data to ensure that your data are not lost.	Make a backup copy of all data that is important to you. If you need more diskettes, contact the operations department.
Use parallel grammatical structures: Parallel grammatical structures indicate the similarity among items in lists and help the reader understand content.	Opening files Saving a document How to delete files	Opening a file Saving a file Deleting a file
Use steps correctly: Novices often intersperse actions and the results of actions when describing a step-by-step process. Steps are always actions.	1. Press the *customer* button. 2. The customer dialogue box will appear. 3. Type the customer ID and press the *submit* button and the customer record will appear.	1. Press the *customer* button. 2. Type the customer ID in the customer dialogue box when it appears. 3. Press the *submit* button to view the customer record for this customer.
Use short paragraphs: Readers of documentation usually quickly scan text to find the information they need, so the text in the middle of long paragraphs is often overlooked. Use numerous separate paragraphs to help readers find information more quickly.		

Source: Adapted from *Writing Software Documentation*, Boston: Aliyn & Bacon, 1998, by T. T. Barker.

FIGURE 12-8
Guidelines for Crafting Documentation Topics

CONCEPTS IN ACTION

12-C Systems for Complex Electrical Systems

Systems integration across platforms and companies grows more complex with time. In a case study from Florida in 2008, an electrical company real-time system detected a minor problem in the power grid and shut down the entire system, plunging over two million people into the dark. The system experts placed the blame on a substation software system that detected the minor fluctuation, but had the ability to immediately shut the entire system down. Although there may be times when such a rapid response is vital (such as the nuclear disasters in Chernobyl Ukraine and Three Mile Island), this was a case where such a response was not warranted.

Questions:

1. Since software controls substation operations, how might a systems analyst approach this problem as a systems project?
2. Are there special considerations that a systems analyst needs to think about when dealing with real-time systems?

command set, but sometimes they require several commands and use terms that do not always appear in the system. A good source for these terms is the use scenarios developed by interface design. (See Chapter 9.)

A fourth, often controversial, source is the set of synonyms for the three sets of items mentioned previously. Users sometimes don't think in terms of the nicely defined terms used by the system. They may try to find information on how to *stop* or *quit* rather than *exit*, or on how to *erase* rather than *delete.* Including synonyms in the index increases the complexity and size of the documentation system but can greatly improve the value of the system to the users.

APPLYING THE CONCEPTS AT TUNE SOURCE

Managing Programming

Three programmers were assigned by Tune Source to develop the three major parts of the Digital Music Download system. The first was the Web interface, both the client side (browser) and the server side. The second, the purchase transaction system, was client–server based. The third was the sales analysis and promotions portion of the system. Programming went smoothly and, despite a few minor problems, according to plan.

Testing

While the programmers were working, Jason—senior systems analyst and project manager for Tune Source's Digital Music Download system—began developing the test plans and user documentation. The test plans for the three components were similar, but slightly more intensive for the Web interface component (Figure 12-9). Unit testing by black-box testing from program specifications was planned for all components. Figure 12-10 shows part of one unit test for the Web interface component.

Integration testing for the Web interface and system management components would encompass all user interface and use scenario tests to ensure that the

Test Stage	Web Interface	System Management	System Interfaces
Unit tests	Black-box tests	Black-box tests	Black-box tests
Integration tests	User interface tests; use scenario tests	User interface tests; use scenario tests	System interface tests
System tests	Requirements tests; security tests; performance tests; usability tests	Requirements tests; security tests	Requirements tests; security tests; performance tests
Acceptance tests	Alpha test; beta test	Alpha test; beta test	Alpha test; beta test

FIGURE 12-9
Tune Source's Test Plan

interface worked properly. The system interface component would undergo system interface tests to ensure that the system performed calculations properly and was capable of exchanging data with external systems used for credit card authorization.

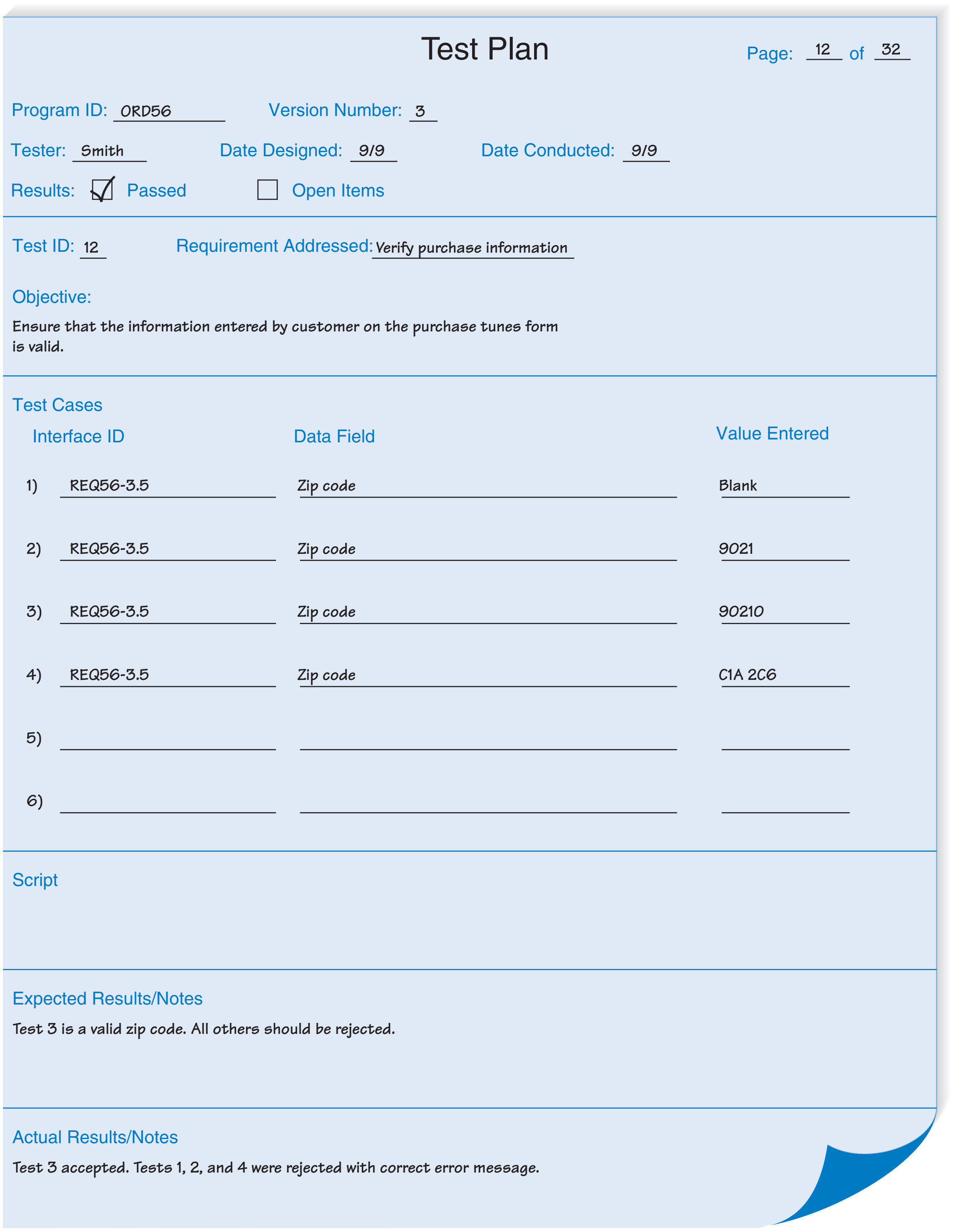
Systems tests are by definition tests of the entire system—all components together. However, not all parts of the system would receive the same level of testing. Requirements tests would be conducted on all parts of the system to ensure that all requirements were met. Security was a critical issue, so the security of all aspects of the system would be tested. Security tests would be developed by Tune Source's infrastructure team, and once the system passed those tests, an external security consulting firm would be hired to attempt to break into the system.

Performance was an important issue for the parts of the system used by the customer (the Web interface and the system interfaces for payment processing), but not as important for the promotions component that would be used by staff, not customers. The customer-facing components would undergo rigorous performance testing to see how many transactions (whether searching or purchasing a download) they could handle before they were unable to provide a response time of 2 seconds or less. Jason also ensured that the architecture design included an upgrade plan so that, as demand on the system increased, there was a clear plan for when and how to increase the processing capability of the system.

Finally, formal usability tests would be conducted on the Web interface portion of the system, with six potential users (both novice and expert Internet users).

Acceptance tests would be conducted in two stages, alpha and beta. Alpha tests would be done during the training of Tune Source's store staff in the use of the in-store kiosks. Carly would work together with Jason to develop a series of tests and training exercises to train staff on how to use the system. They would then load the real music data into the system. These same staff and other Tune Source staff members would also pretend to be customers and test the Web interface.

Beta testing would begin by "going live" with the in-store kiosks. In-store shoppers would be offered a free download in return for evaluating the system. Then, the Web site would go live, but announced only to Tune Source employees. As an incentive to try the Web site, employees would be offered five free downloads from the Web site. The site would also have a prominent button on every screen that would enable employees to e-mail comments to the project team, and the announcement would encourage employees to report problems, suggestions,



Test Plan

Page: 12 of 32

Program ID: ORD56 Version Number: 3

Tester: Smith Date Designed: 9/9 Date Conducted: 9/9

Results: [✓ Passed] ☐ Open Items

Test ID: 12 Requirement Addressed: Verify purchase information

Objective:

Ensure that the information entered by customer on the purchase tunes form is valid.

Test Cases

	Interface ID	Data Field	Value Entered
1)	REQ56-3.5	Zip code	Blank
2)	REQ56-3.5	Zip code	9021
3)	REQ56-3.5	Zip code	90210
4)	REQ56-3.5	Zip code	C1A 2C6
5)			
6)			

Script

Expected Results/Notes

Test 3 is a valid zip code. All others should be rejected.

Actual Results/Notes

Test 3 accepted. Tests 1, 2, and 4 were rejected with correct error message.

FIGURE 12-10
Tune Source Unit Test Plan Example

and compliments to the project team. After one month, assuming that all went well, the beta test would be completed and the site would be linked to the main Web site and advertised to the general public.

Developing User Documentation

There were three types of documentation (reference documents, procedures manuals, and tutorials) that could be produced for the Web interface and the promotion component. Since the number of Tune Source staff using the promotion component would be small, Jason decided to produce only the reference documentation (an online help system). He believed that an intensive training program and a one-month beta-test period would be sufficient without tutorials and formal procedure manuals. Likewise, he felt that the process of purchasing tunes and the user interface itself were simple enough not to require a tutorial on the Web—a help system would be sufficient, and a procedure manual didn't make sense.

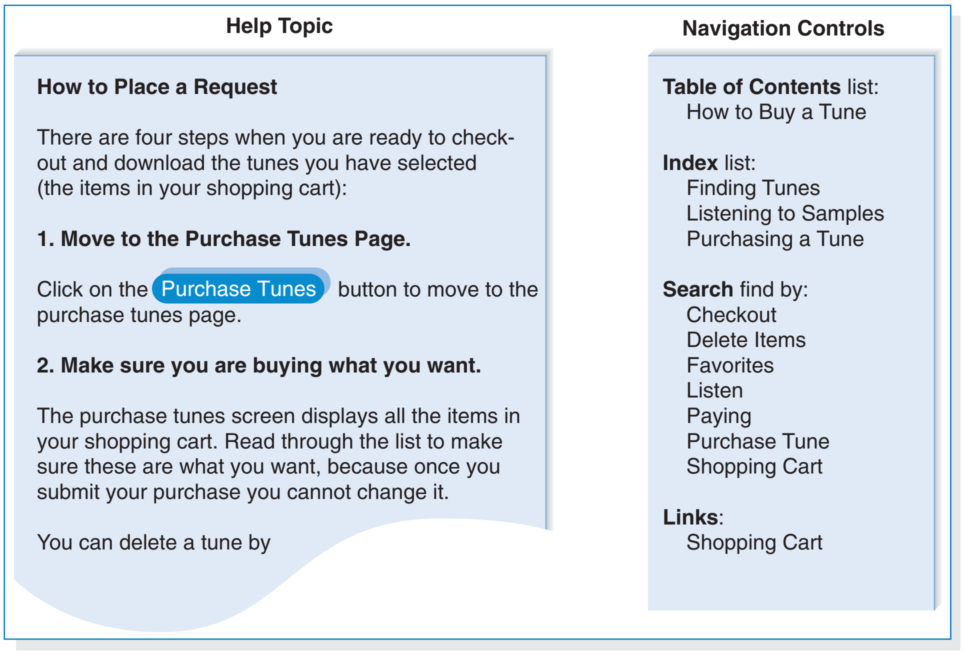
Jason decided that the reference documents for both the Web interface and promotion components would contain help topics for user tasks, commands, and definitions. He also decided that the documentation component would contain four types of navigation controls: a table of contents, an index, a finder, and links to definitions. He did not think that the system was complex enough to benefit from a search agent.

After these decisions were made, Jason assigned the development of the reference documents to a technical writer assigned to the project team. Figure 12-11 shows examples of the topics the writer developed. The tasks and commands were taken directly from the interface design. The list of definitions was put together, once the tasks and commands were developed, on the basis of the writer's experience in understanding what terms might be confusing to the user.

Once the topic list was developed, the technical writer then began writing the topics themselves and the navigation controls to access. Figure 12-12 shows an example of one topic taken from the task list: how to place a request. This topic presents a brief description of what it is and then leads the user through the step-by-step process needed to complete the task. The topic also lists the navigation controls that will be used to find the topic, in terms of the table of contents entries, index entries, and search entries. It also lists what words in the topic itself will have links to other topics (e.g., shopping cart).

Tasks	Commands	Terms
Find tune.	Find	Tune
Add a tune to my shopping cart.	Browse	Artist
Add a favorite.	Quick search	Genre
Checkout.	Full search	Special deals
What's in my shopping cart?		Cart
		Shopping cart

FIGURE 12-11
Sample Help Topics for Tune Source



Help Topic

How to Place a Request

There are four steps when you are ready to check-out and download the tunes you have selected (the items in your shopping cart):

1. Move to the Purchase Tunes Page.

Click on the Purchase Tunes button to move to the purchase tunes page.

2. Make sure you are buying what you want.

The purchase tunes screen displays all the items in your shopping cart. Read through the list to make sure these are what you want, because once you submit your purchase you cannot change it.

You can delete a tune by

Navigation Controls

Table of Contents list:
How to Buy a Tune

Index list:
Finding Tunes
Listening to Samples
Purchasing a Tune

Search find by:
Checkout
Delete Items
Favorites
Listen
Paying
Purchase Tune
Shopping Cart

Links:
Shopping Cart

FIGURE 12-12
Example Documentation Topic for Tune Source

SUMMARY

Managing Programming

Programming is done by programmers, so systems analysts have other responsibilities during this stage. The project manager, however, is usually very busy. The first step is to assign tasks to the programmers—ideally, the fewest possible to complete the project, because coordination problems increase as the size of the programming team increases. Coordination can be improved by having regular meetings, ensuring that standards are followed, implementing change control, and using computer-aided software engineering (CASE) tools effectively. One of the key functions of the project manager is to manage the schedule and adjust it for delays. Two common causes of delays are scope creep and minor slippages that go unnoticed.

Testing

Tests must be carefully planned because the cost of fixing one major bug after the system is installed can easily exceed the annual salary of a programmer. A test plan contains several tests that examine different aspects of the system. A test, in turn, specifies several test cases that will be examined by the testers. A unit test examines a module or program within the system; test cases come from the program specifications or the program code itself. An integration test examines how well several modules work together; test cases come from the interface design, use scenarios, and the physical data flow diagrams (DFDs). A system test examines the system as a whole and is broader than the unit and integration tests; test cases come from the system design, the infrastructure design, the unit tests, and the integration. Acceptance testing is done by the users to determine whether the system is acceptable to them; it draws on the system test plans (alpha testing) and the real work the users perform (beta testing).

Documentation

Documentation, both user documentation and system documentation, is moving away from paper-based documents to online documentation. There are three types of user documentation: Reference documents are designed to be used when the user needs to learn how to perform a specific function (e.g., an online help system); procedures manuals describe how to perform business tasks; and tutorials teach people how to use the system. Documentation navigation controls (e.g., a table of contents, index, a "find" function, intelligent agents, or links between pages) enable users to find documentation topics (e.g., how to perform a function, how to use an interface command, an explanation of a term).

KEY TERMS

Acceptance test
Alpha test
Beta test
Black-box testing
Change control
Construction
Data flow testing
Documentation navigation control
Documentation testing
Documentation topic
Hardcoded
Integration test
Performance testing
Procedures manual
Program log
Reference document
Requirements testing
Scope creep
Security testing
Stub
System documentation
System interface testing
System test
Test case
Test plan
Tutorial
Unit test
Usability testing
Use scenario testing
User documentation
User interface testing
White-box testing

QUESTIONS

1. Discuss the issues the project manager must consider when assigning programming tasks to the programmers.
2. If the project manager feels that programming is falling behind schedule, should more programmers be assigned to the project? Why or why not?
3. Describe the typical way that project managers organize the programmers' work storage areas. Why is this approach useful?
4. What is meant by change control? How is it helpful to the programming effort?
5. Discuss why testing is so essential to the development of the new system.
6. Explain how a test case relates to a test plan.
7. What is the primary goal of unit testing?
8. How are test cases developed for unit tests?
9. What is the primary goal of integration testing?
10. Describe the four approaches to integration testing.
11. How are the test cases developed for integration tests?
12. Compare and contrast black-box testing and white-box testing.
13. Compare and contrast system testing and acceptance testing.
14. Describe the five approaches to systems testing.
15. Discuss the role users play in testing.
16. What is the difference between alpha testing and beta testing?
17. Explain the difference between user documentation and system documentation.
18. What are the reasons underlying the popularity of online documentation?
19. Are there any limitations to online documentation? Explain.
20. Distinguish between these types of user documentation: reference documents, procedures manuals, and tutorials.
21. Describe the five types of documentation navigation controls.
22. What are the commonly used sources of documentation topics? Which is the most important? Why?
23. What are the commonly used sources of documentation navigation controls? Which is the most important? Why?
24. What do you think are three common mistakes made by novice systems analysts during programming and testing?

25. What do you think are three common mistakes made by novice systems analysts in preparing user documentation?
26. In our experience, documentation is left to the very end of most projects. Why do you think this happens? How could it be avoided?
27. In our experience, few organizations perform as thorough testing as they should. Why do you think this happens? How could it be avoided?
28. Create several guidelines for developing good documentation. Hint: Think about behaviors that might lead to developing poor documentation.

EXERCISES

A. Develop a unit test plan for the calculator program in Windows (or a similar program for the Mac or UNIX).
B. Develop a unit test plan for a Web site that enables you to perform some function (e.g., make travel reservations, order books).
C. If the registration system at your university does not have a good online help system, develop one for one screen of the user interface.
D. Examine and prepare a report on the online help system for the calculator program in Windows (or a similar program for the Mac or Unix). (You may be surprised at the amount of help that is available for such a simple program).
E. Compare and contrast the online help resources at two different Web sites that enable you to perform the same function (e.g., make travel reservations, order books).

MINICASES

1. A new systems development project is Pete's first experience as a project manager, and he has led his team successfully to the programming phase of the project. The project has not always gone smoothly, and Pete has made a few mistakes, but he is generally pleased with the progress of his team and the quality of the system being developed. Now that programming has begun, Pete has been hoping for a little break in the hectic pace of his workday.

Prior to beginning programming. Pete recognized that the time estimates made earlier in the project were too optimistic. However, he was firmly committed to meeting the project deadline because of his desire for his first project to be a success. In anticipation of this time-pressure problem, Pete arranged with the human resources department to bring in two new college graduates and two college interns to beef up the programming staff. Pete would have liked to find some staff with more experience, but the budget was too tight and he was committed to keeping the project budget under control.

Pete made his programming assignments, and work on the programs began about two weeks ago. Now, Pete has started to hear some rumbles from the programming team leaders that may signal trouble. It seems that the programmers have reported several instances where they wrote programs, only to be unable to find them when they went to test them. Also, several programmers have opened programs that they had written, only to find that someone had changed portions of their programs without their knowledge.

a. Is the programming phase of a project a time for the project manager to relax? Why or why not?
b. What problems can you identify in this situation?
c. What advice do you have for the project manager?
d. How likely does it seem that Pete will achieve his desired goals of being on time and within budget if nothing is done?

2. The systems analysts are developing the test plan for the user interface for the Holiday Travel Vehicles system. As the salespeople are entering a sales invoice into the system, they will be able to either enter an option code into a text box or select an option code from a drop-down list. A combo box was used to implement this, since it was felt that the salespeople would quickly become familiar with the most common option codes and would prefer entering them directly to speed up the entry process.

It is now time to develop the test for validating the option code field during data entry. If the customer did not request any dealer-installed options for the vehicle, the salesperson should enter "none"; the field should not be blank. The valid option codes are four-character alphabetic codes and should be matched against a list of valid codes.

Prepare a test plan for the test of the option code field during data entry.